Domain-Theory Case-Study based on Deep-Space One

Automatic Software Code Generator

Deliverable A2

Nicolas F. Rouquette

1 DS1 Background

The domain modeling of monitor telemetry is a particularly interesting case of where modeling assumptions were made early in the project about the relevant information necessary to specify each monitor telemetry measurement. We had initially concentrated on modeling the functionality necessary to fulfill the basic fault-protection requirements. Later, when the telemetry issue was re-visited, we found that additional modeling information had to be specified in order to fully characterize what kind of telemetry to produce and when. We use this case to show illustrate how we built the domain-theory case study for DS1.

2 Basic Descriptions of Monitors

The initial modeling of monitors focused on capturing the basic functionality. For monitors, this meant defining the set of parameters and variables needed in the computation of the monitor. Here is an example of how parameters are defined:

```
begin parameters

double delta_x        = 0.005;
               // (32bit floating point: range=[0.0;100.0]) amount of padding added to
                   the x-axis ACS deadbands to calculate the min/max threshold bounds for
                   the F component of error(x) & error rate(x)
…

double tolerated_time = 3.0;
               // (32bit floating point: range=[0.0;10000.0]) Tolerated time outside
                   deadbands. This implies slope of g axis.

int  transition_time = 60;
               // (32bit signed integer: range=[0;2^30]) Delay time used to temporarily
                   disable the control error monitor when ACS changes the deadbands.

int confidence        = 5;
               // (32bit signed integer: range=[0;2^30]) minimum number of cycles of data
                   within min/max range to define a "green" threshold state.
int persistence       = 3;
               // (32bit signed integer: range=[0;2^30]) minimum number of cycles of data
                   outside min/max range to define a "red" threshold state.
int decay      = 5;
               // (32bit signed integer: range=[0;2^30]) minimum number of cycles of no-
                   data to reinitialize the threshold tracking logic.
end parameters
```

Here is an example of monitor variables (they were called "elements" for historical reasons):

```
begin elements
int delay = mon_control_error_mon_object.parms.transition_time;
               // (32bit signed integer) count of 'done' ticks remaining before enabling
                   the monitor

/* zero will force quick alarms if not changed by acs */
double deadband_x = 0.0; :no-reinit
               // (32bit floating point) current deadband on x axis (takes account any
                   applicable tightening delay)
…
double error_x = 0.0;
               // (32bit floating point) current error for x axis
…
```

```
double error_rate_x = 0.0;
               // (32bit floating point) current error rate for x axis
…
end elements
```

The domain specification language includes the notion of a component to define a chunk of general-purpose functionality into a reusable unit. The following example shows how a threshold component is instantiated in the control-error monitor to detect when the component input becomes persistently too large (i.e., above the maximum of **`deadband_x + delta_x`**) or too small (i.e., above the minimum of: **`-deadband_x - delta_x`**):

```
begin components
mon_threshold_component f_x( -(deadband_x+delta_x), deadband_x+delta_x, confidence,
                                    persistence, decay );
               // f component of the phase plane defined by: error(x), rate_error(x)
…
end components
```

The declarative nature of such descriptions is of fundamental importance for decoupling the code generation machinery (domain-independent knowledge) from the domain-specific knowledge of a particular monitor design and requirements. However, the specification language shown above through examples has a fair amount of implicit knowledge attached to it. While directives such as **`begin`** and **`end`** can be reasonably attributed to the notion of grouping declarations into a unit, other directives are much more tied to the particular application (e.g., **`parameter`**, **`variable`**, **`double`**, **`updates`**,…) because the code generator uses them to define a *template language* for referring to any attribute of a specification. For example, here is a excerpt of the template used to define the C data structure that holds the run-time data of a monitor:

```
typedef struct {
#forall elements
    /* @<element_description> */
    @<element_type> @<element_name>;
#end elements
#forall components
    /* @<component_description> */
    @<component_type>_struct_t @<component_name>@<component_dim>;
#end components
} $<name>_dynamic_state_t;
```

In the example of the control-error monitor, the above template produces the following code:

```
typedef struct
  {
    /* (32bit signed integer) count of 'done' ticks remaining before enabling the monitor
        */
    int delay;

    /* (32bit floating point) current deadband on x axis (takes account any applicable
        tightening delay) */
    double deadband_x;
…
    /* (32bit floating point) current error for x axis */
    double error_x;
…
    /* (32bit floating point) current error rate for x axis */
    double error_rate_x;
…
   /* f component of the phase plane defined by: error(x),
      rate_error(x) */
    mon_threshold_component_struct_t f_x;
…
} mon_control_error_mon_dynamic_state_t;
```

This type of code-generation technology is based on a simple text substitution mechanism. In the above example, the keyword: **`@<element_name>`** is replaced by the names of the current element. The notion of the

current element is controlled by the context of the #forall elements … #end elements loop which iterates over each element of the current monitor. In this case, the notion of the current monitor context is controlled by the fact that the template is applied to the control-error monitor specification.
Text-substitution mechanisms have some limitations, some of which are more difficult to anticipate than others as was the case for the monitor telemetry in DS1.

3 Limitations of Text-Substitution for Code Generation

In DS1, the modeling of monitor telemetry was not a high priority task and was consequently postponed until the basic monitor functionality was properly generated. The priority was justified on the basis that designing the core monitor functionality for processing arbitrary data inputs and for detecting arbitrary signatures of anomaly symptoms was much more complex than the passive role telemetry has with respect to providing ground visibility to the monitor internal variables. However, additional requirements were introduced: because the spacecraft will spend a lot of time without ground visibility, summarizing behavior with water marks and statistical measures became much more important. This implied adding to the monitor dynamic state enough variables to compute the behavior summarization and update the telemetry statistics.
The following example shows how telemetry statistics and water marks were added to the control error monitor specification. The first specification clause declares that state statistics will be computed for the current state of the 'f_x' component and its possible values. The second clause declares that the local maxima of the input of the threshold component 'f_x' will be tracked when the input value will be within the min/max values.

```
begin telemetry
…
F_X_STATE:     STATE(f_x.current_state.vars.state,mon_threshold_states);
…
F_X_THRESHOLD: HIGH_MARK( \
   f_x.current_state.vars.input, \
   (f_x.current_state.parms.Max> f_x.current_state.vars.input && \
    f_x.current_state.vars.input > f_x.current_state.parms.Min), \
   f_x.current_state.parms.Min)
…
end telemetry
```

The actual definition of the F_X_STATE telemetry and associated variables necessary to compute it depends on the number of different states that `f_x.current_state.vars.state` can have. This information can be easily obtained from the public interface header files where the following definition of `mon_threshold_states` can be found:

```
typedef enum {
    thrUnknown,
    thrLow,
    thrMaybeLow,
    thrNominal,
    thrMaybeNominal,
    thrHigh,
    thrMaybeHigh
} mon_threshold_states;
```

The code generator uses this information to calculate, for example, how many bits are necessary to calculate the statistics of how frequently each state occurs and how often state changes occur. This mechanism cannot be cleanly described in terms of a simple syntactic replacement rule. Indeed, the DS1 template is succinct because the expansion hides all of the internal processing necessary to perform type information lookup and subsequent processing:

```
typedef struct {
#forall elements
    /* Telemetry for element: @<element_name> */
    @<declare_telemetry(@<element_name>)>;
#end elements
#forall components
```

```
    /* Telemetry for component: @<component_name> */
    @<declare_telemetry(@<component_name>)>;
#end components
} $<name>_telemetry
```

… but the expansion of this template illustrates that the code-generation is much more complex than piecing together a few text fields:

```
typedef struct
  {
…
    /* Telemetry for component: f_x */
    mon_state_t F_X_STATE_previous_state;
    mon_length_t F_X_STATE_episode_length;
    mon_length_t F_X_STATE_episode_history[7];
    mon_length_t F_X_STATE_cumulative_history[7];
…
    int F_X_THRESHOLD_high_mark_state;
    double F_X_THRESHOLD_high_mark_current;
    double F_X_THRESHOLD_high_mark_achieved;
    int F_X_THRESHOLD_low_mark_state;
    double F_X_THRESHOLD_low_mark_current;
    double F_X_THRESHOLD_low_mark_achieved;
…
} mon_control_error_mon_telemetry;
```

The keyword: **@<declare_telemetry>** hides a lot of code-generation work. While this would be acceptable if it were an isolated instance, it raises more concerns when other code-generation mechanisms for related purposes need to hide a similarly complex logic. The keyword: **@<add_telemetry>** is one such example where for the same telemetry definition, the generated code becomes:

```
mon_calculate_state_eha
  (MON_CONTROL_ERROR_MON_EHA_F_X_STATE,
  (unsigned char)
   mon_control_error_mon_object.functional_state.f_x.current_state.vars.state,
  &mon_control_error_mon_object.telemetry_state.F_X_STATE_previous_state,
  MON_CONTROL_ERROR_MON_EHA_F_X_STATE_EPISODIC_SUMMARY,
  &mon_control_error_mon_object.telemetry_state.F_X_STATE_episode_length,
  MON_CONTROL_ERROR_MON_EHA_F_X_STATE_EPISODIC_HISTORY,
  &mon_control_error_mon_object.telemetry_state.F_X_STATE_episode_history[0],
  MON_CONTROL_ERROR_MON_EHA_F_X_STATE_CUMULATIVE_HISTORY,
  &mon_control_error_mon_object.telemetry_state.F_X_STATE_cumulative_history[0]);
```

and:

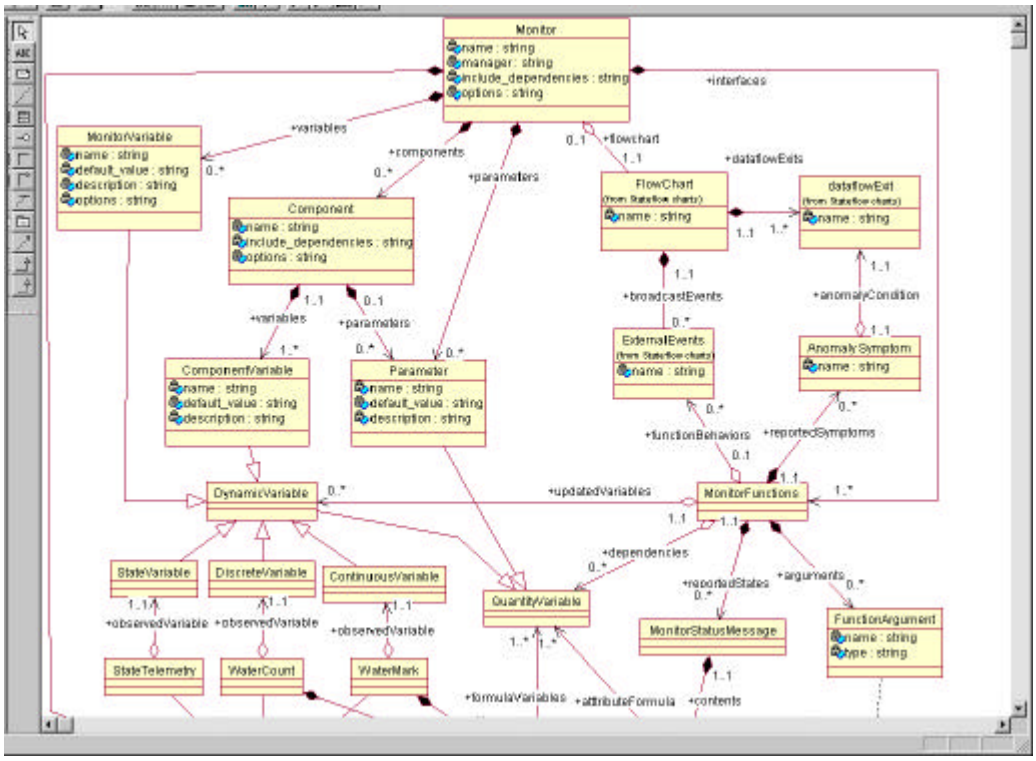
```
mon_calculate_high_mark_eha
 (mon_control_error_mon_object.functional_state.f_x.current_state.parms.Max >
   mon_control_error_mon_object.functional_state.f_x.current_state.vars.input &&
   mon_control_error_mon_object.functional_state.f_x.current_state.vars.input >
   mon_control_error_mon_object.functional_state.f_x.current_state.parms.Min,
 mon_control_error_mon_object.functional_state.f_x.current_state.vars.input,
 &mon_control_error_mon_object.telemetry_state.F_X_THRESHOLD_high_mark_state,
 &mon_control_error_mon_object.telemetry_state.F_X_THRESHOLD_high_mark_current,
 &mon_control_error_mon_object.telemetry_state.F_X_THRESHOLD_high_mark_achieved,
 MON_CONTROL_ERROR_MON_EHA_HIGH_WATER_MARK_OF_F_X_THRESHOLD);
```

4 The Domain-Theory Approach

The key technical advance we are pursuing to address the problems illustrated above consists in storing domain-specific knowledge into a database. This knowledge can be separated in two categories: 1) the design specifications of what to build (e.g., the control-error monitor specification of § Sec. 2 and § Sec. 3) and 2) the nature and type information about software entities in the domain (e.g., the type information about **mon_threshold_states**.)

5 Models of Design Information

To construct the database of design information, we need to first define a database *schema*. Such a schema is a model in Rational Rose where each unit of information relevant for the purpose of writing a domain-specific software specification will be explicitly mapped into a database table. The class diagram showing the different concepts relevant to specifying a monitor in the DS1 domain is partially shown below:

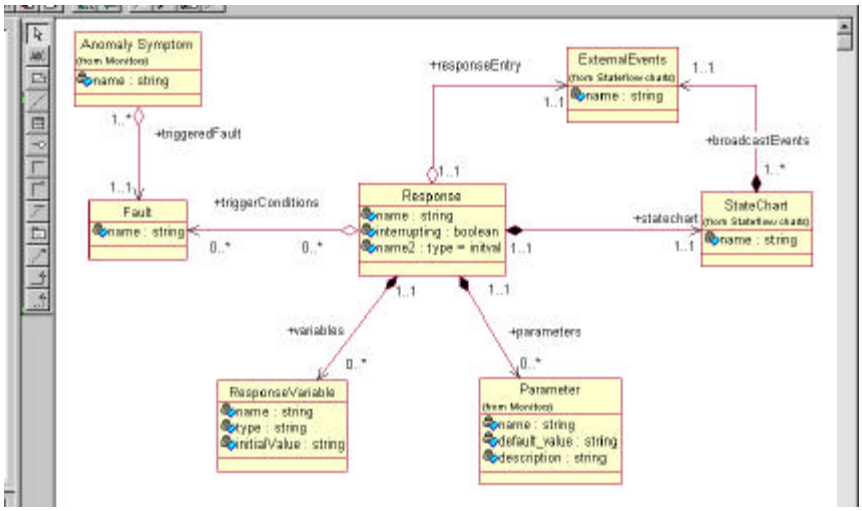


This diagram captures the relationships there exists amongst the various parts of a monitor specification file. For the example of § sec. 2, 'control-error' is an instance of a monitor specification which includes a parameter specification with the following properties: name='`delta_x`', default_value='`0.005`', description='`(32bit floating point: range=[0.0;100.0]) amount of padding added to the x-axis ACS deadbands to calculate the min/max threshold bounds for the F component of error(x) & error rate(x)`'. Each entity specified in § sec. 2 maps to an instance of a specification class in the above diagram.

The specification diagram also includes additional classes such as 'DynamicVariable' and 'QuantityVariable'. Such classes introduce an abstraction of the specification classes to enable other specification relationships to be described more precisely. For example, the specification class for 'MonitorFunction' has an 'updatedVariables' aggregate relationship with 'DynamicVariable' to indicate that among all 'QuantityVariables', only the 'DynamicVariables' can be updated by a monitor function while that function can refer to and therefore depend on any 'QuantityVariable' defined in the specification.

A number of constraints can be defined to define a declarative criteria of validity for specifications. The previous example shows a simple membership constraint that is easily represented in the Rose model. For complex validity constraints, a visual representation can be more confusing than a textual description in a formal language. For such cases, the validity constraints on the specification model have to be captured outside the Rose model (this is a known issue to be addressed.)

A similar diagram exists to describe the design information necessary for specifying fault-protection responses:



Rational Rose generates automatically from such diagrams SQL database schemas with which we can create a specification database for recording all instances of the fault-protection monitors and response designs in the domain. The delivery A3 will describe the full model of fault protection and the delivery A4 will describe how these kinds of domain theory models are used in a database-driven code-generation.

6 Contributors for this deliverable

Nicolas Rouquette
Julia Dunphy